Introduction to Computing Part II

In this lab you will write a subroutine, plot your output and learn how to automatically regenerate the plot.

Last week you logged in, created a subdirectory, used an editor to write a program, compiled the program and ran it. The commands used were

`mkdir`	— make directories
`ls`	— list directory contents
`cd`	— change working directory
`pwd`	— print name of current/working directory
`gedit`	— text editor for the GNOME Desktop
`gcc`	— GNU project C and C++ compiler

We will use these commands again in this lab while introducing new commands as needed.

Please log into your workstation. Hopefully all problems with UNR netids and passwords have been resolved since last time. If you still can't log in, this is an emergency. Please contact me at `ejolson@unr.edu` to set up an appointment to get this problem addressed.

In order to keep our files organized, let's start by making a new subdirectory for the work we do today. Please open a terminal window and enter the following commands.

```
netid@hostname:~$ cd
netid@hostname:~$ ls
lab01
netid@hostname:~$ mkdir lab02
netid@hostname:~$ ls
lab01  lab02
netid@hostname:~$ cd lab02
netid@hostname:~/lab02$ pwd
/nfs/home/netid/lab02
```

Now copy the program from last week into the new directory.

```
netid@hostname:~/lab02$ cp ../lab01/main.c .
netid@hostname:~/lab02$ ls
main.c
```

The copy command always has a source followed by destination. In our case, the destination is `.` which denotes the current working directory.

Creating a Makefile

Before proceeding, let's check that the C source code in `main.c` from last week still works. We will do this by compiling the code and then running the executable to check that the output is as expected. As a program is developed it will be compiled many times. We will use the `make` utility to automate this task. This utility relies on a file named `Makefile` that we will now create using `gedit` with

```
netid@hostname:~/lab02$ gedit Makefile &
```

Please remember to include the trailing `&` on the above line. This will allow us to use the terminal window while the editor is still open.

Before proceding it is unfortunately necessary to have a discussion on the difference between a tab character and a space character in the ASCII character set. The tab is ASCII code 9 whereas the space is ASCII code 32. Although they look the same, these two characters have different meanings in a `Makefile`. Even worse, some editors automatically convert spaces to tabs or tabs to spaces. To make sure `gedit` is configured to process tabs properly, open

Edit $\rightarrow$ Preferences $\rightarrow$ Editor

and make sure Insert spaces instead of tabs is *not* selected. You may also want to change Tab width to 4, which is the width used in the original book by Kernighan and Ritchie on C programming.

Now, edit `Makefile` so it contains

```
main: main.c
    gcc -O2 -std=gnu99 -o main main.c -lm
```

Here, the whitespace that appears before `gcc` must be a tab and not a series of spaces. Note that if you set Tab width to 8 in `gedit`, the same file would look like

```
main: main.c
        gcc -O2 -std=gnu99 -o main main.c -lm
```

The compiler command is longer than the one we used last week. The additional options are described below.

`-O2` Tells the compiler to perform level 2 optimizations when generating the executable. This is often important for numerical codes. Data flow analysis eliminates unnecessary computation while register mapping and vectorization yield the efficient sequencing of hardware floating point instructions.

`-o main` Specifies that the name of the resulting executable will be `main` rather than the default `a.out`. If you want to impress your neighbor with your historical knowledge, please take a moment to explain what the `a` in `a.out` originally signified.

It should now be possible to compile the program by typing

```
netid@hostname:~/lab02$ make
gcc -O2 -std=gnu99 -o main main.c -lm
```

If you get the error

```
netid@hostname:~/lab02$ make
cc     main.c   -o main
main.c: In function main:
main.c:5:5: error: for loop initial declarations are only allo
wed in C99 mode
main.c:5:5: note: use option -std=c99 or -std=gnu99 to compile
your code
make: *** [main] Error 1
```

then you did not take care of the tab character as requested. Please delete the spaces before `gcc` in `Makefile` and insert a proper tab character using the tab key on the computer keyboard. Save the file and try again.

The resulting execuatable should be called `main` and you can run it using the following sequence of commands.

```
netid@hostname:~/lab02$ ls
Makefile  main  main.c
netid@hostname:~/lab02$ ./main
    x               erf(x)
  0.1   1.124629160183e-01
  0.2   2.227025892105e-01
  0.3   3.286267594591e-01
  0.4   4.283923550467e-01
  0.5   5.204998778130e-01
  0.6   6.038560908479e-01
  0.7   6.778011938374e-01
  0.8   7.421009647077e-01
  0.9   7.969082124228e-01
    1   8.427007929497e-01
```

Some Mathematical Simplifications

In the next section we will code a subroutine to compute the sum

$$\frac{2}{\sqrt{\pi}}\sum_{k=0}^{n}\frac{(-1)^k}{k!(2k+1)}x^{2k+1}$$

where n is a large positive integer. It is tempting to translate this sum directly into a loop, however, this does not result in an efficient or accurate calculation. Instead define

$$a_k=\frac{(-1)^k}{k!(2k+1)}\qquad\text{and observe that}\qquad\frac{a_k}{a_{k-1}}=-\frac{2k-1}{k(2k+1)}$$

so that

$$a_0=1,\quad \frac{a_1}{a_0}=\frac{-1}{3},\quad \frac{a_2}{a_1}=\frac{-3}{10}\quad\text{and}\quad\frac{a_3}{a_2}=\frac{-5}{21}.$$

Therefore

$$\begin{aligned}\sum_{k=0}^{3}\frac{(-1)^k}{k!(2k+1)}x^{2k+1}&=a_0x+a_1x^3+a_2x^5+a_3x^7\\&=a_0x+a_1x^3+a_2x^5\Big(1+\frac{a_3}{a_2}x^2\Big)\\&=a_0x+a_1x^3\Big(1+\frac{a_2}{a_1}x^2\Big(1+\frac{a_3}{a_2}x^2\Big)\Big)\\&=a_0x\Big(1+\frac{a_1}{a_0}x^2\Big(1+\frac{a_2}{a_1}x^2\Big(1+\frac{a_3}{a_2}x^2\Big)\Big)\Big)\\&=x\Big(1-\frac{1}{3}x^2\Big(1-\frac{3}{10}x^2\Big(1-\frac{5}{21}x^2\Big)\Big)\Big).\end{aligned}$$

The above pattern can evidently be extended for any length sum. Working from the inner to the outer parenthesis we obtain that

$$\sum_{k=0}^{n}\frac{(-1)^k}{k!(2k+1)}x^{2k+1}=x\sigma_1$$

where

$$\sigma_{n+1}=1\qquad\text{and}\qquad\sigma_k=1-\frac{a_k}{a_{k-1}}x^2\sigma_{k+1}$$

for $k=1,\ldots,n$. We now write C code to implement this calculation.

Writing the Subroutine

The C code for a subroutine `myerf` which computes our sum approximation to the standard error function may be written as

```
double myerf(double x,int n){
    double xx=x*x, sigma=1;
    for(int k=n;k>=1;k--){
        sigma=1-(2*k-1)*xx*sigma/(k*(2*k+1));
    }
    return x*sigma*2/sqrt(M_PI);
}
```

This routine contains a loop which evaluates the σ_k in reverse order to work from the inner to the outer parenthesis in our derivation. Note also the division by `sqrt(M_PI)` at the end. The constant `M_PI` is defined in `math.h` to be as close to the mathematical constant π as machine precision allows. Let's add this routine to our program.

In order to load `main.c` into your `gedit` session type the following in the terminal window.

```
netid@hostname:~/lab02$ gedit main.c &
```

After a certain amount of typing the final program should read as

```
#include <stdio.h>
#include <math.h>
double myerf(double x,int n){
    double xx=x*x, sigma=1;
    for(int k=n;k>=1;k--){
        sigma=1-(2*k-1)*xx*sigma/(k*(2*k+1));
    }
    return x*sigma*2/sqrt(M_PI);
}
int main(){
    printf("%5s %20s %20s\n","x","erf(x)","myerf(x)");
    for(int i=0;i<=10;i++){
        double x=i/10.0;
        printf("%5g %20.12e %20.12e\n", x, erf(x), myerf(x,7));
    }
    return 0;
}
```

Type `make` to compile it. If there are errors please fix them until an executable is generated. The output of the program should be

```
    x               erf(x)             myerf(x)
    0   0.000000000000e+00   0.000000000000e+00
  0.1   1.124629160183e-01   1.124629160183e-01
  0.2   2.227025892105e-01   2.227025892105e-01
  0.3   3.286267594591e-01   3.286267594591e-01
  0.4   4.283923550467e-01   4.283923550464e-01
  0.5   5.204998778130e-01   5.204998778008e-01
  0.6   6.038560908479e-01   6.038560905789e-01
  0.7   6.778011938374e-01   6.778011901864e-01
  0.8   7.421009647077e-01   7.421009298672e-01
  0.9   7.969082124228e-01   7.969079584713e-01
    1   8.427007929497e-01   8.426992967307e-01
```

Plotting the Output

The plotting program `gnuplot` was created in 1986 by Williams and Kelley for producing publication quality graphics. The program is open source and available on almost all types of computers; however, despite it's name it is not part of the GNU project. We first modify line 11 of our program to prefix the table heading with a `#` so that `gnuplot` will skip that line when reading the data.

```
11      printf("#%4s %20s %20s\n","x","erf(x)","myerf(x)");
```

To compile, execute and store the output into the file `myerf.dat` type the following into your terminal window.

```
netid@hostname:~/lab02$ make
gcc -O2 -o main -std=gnu99 main.c -lm
netid@hostname:~/lab02$ ./main >myerf.dat
```

Now start `gnuplot` in interactive mode.

```
netid@hostname:~/lab02$ gnuplot

        G N U P L O T
        Version 4.6 patchlevel 0    last modified 2012-03-04
        Build System: Linux i686

        Copyright (C) 1986-1993, 1998, 2004, 2007-2012
        Thomas Williams, Colin Kelley and many others

        gnuplot home:     http://www.gnuplot.info
        faq, bugs, etc:   type "help FAQ"
        immediate help:   type "help"  (plot window: hit 'h')
```

```
Terminal type set to 'wxt'
gnuplot> _
```

The prompt `gnuplot>` means that program is waiting for a plotting command. To see a plot of the data for `myerf` type the following:

```
gnuplot> plot "myerf.dat" using 1:3
```

This plots the first and third columns in the `myerf.dat` file.

We'll finish by creating a plot script that superimposes the data points for `myerf` on a line graph of `erf` and writes the output as an encapsulated postscript file which can be included in a report or displayed on the screen. Exit the gnuplot program to get back to the terminal prompt.

```
gnuplot> exit
netid@hostname:~/lab02$ _
```

Use `gedit` to create the file `myerf.plt` containing the lines

```
1 set terminal postscript eps font "Courier-Bold"
2 set output "myerf.eps"
3 set key bottom
4 set size 0.7,0.7
5 plot "myerf.dat" using 1:3 pt 7 ti "myerf", \
6     "" using 1:2 with lines ti "erf"
```

This script contains the same plotting command as before along with some additional options for better formatting.

Line 1: The output is encapsulated postscript.

2: Specify where the output will be written.

3: Place the key at the bottom so it is out of the way.

4: Setting the graph size smaller has the relative effect of making the text and labels appear larger.

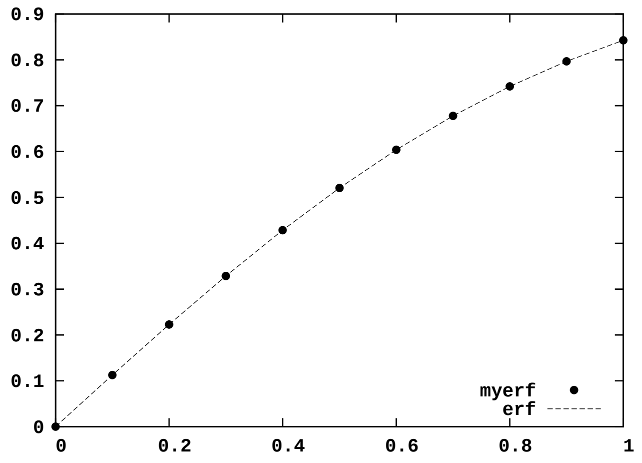
5: The option `pt 7` means use point type 7 which is a filled-in circle. The line ends with `\` to continue plotting on the next line.

6: The filename `""` indicates to read the file `myerf.dat` again for plotting different data.

Process the script and view the resulting graph using the commands

```
netid@hostname:~/lab02$ gnuplot <myerf.plt
netid@hostname:~/lab02$ evince myerf.eps &
```

Again pay attention to the & at then end of second command. The plot should look like



Notice that the data points for `myerf` visually lie on the line representing the built-in function `erf`.

Automating Everything

Some of you may have used integrated development environments that allow the editing, debugging and execution of a program with the click of a mouse. You may also have experience with interactive plotting facilities that allow the visual appearance of a plot to be modified with a mouse. Mouse driven programs can be fun, but they are often difficult to integrate together as components in a larger automated system. The tools we have used are not mouse driven; however, they are easy to automate.

In this last section we add additional rules to our `Makefile` to automate the generation of the final plot. We will set things up so that

1. A new plot `myerf.eps` is generated every time `myerf.plt` is changed or `myerf.dat` is updated.
2. The output file `myerf.dat` is updated every time a new executable `main` is generated.
3. A new executable `main` is generated every time the program `main.c` is changed.

Our `Makefile` already contains a rule to compile the program `main.c`. We will add two additional rules to the file.

```
1 myerf.eps: myerf.plt myerf.dat
2 	gnuplot <myerf.plt
3
4 myerf.dat: main
5 	./main >myerf.dat
6
7 main: main.c
8 	gcc -O2 -o main -std=gnu99 main.c -lm
```

Remember there is a tab character at the beginning of lines 2, 5 and 8 above. Although a sequence of spaces visually look the same as a tab character, the `make` program interprets them differently.

Each rule is written in two parts. The first part contains a colon and the second part begins with a tab character. Just before the colon is the target, which is what the rule will make. Right after the colon is the list of files needed to make the target. The line beginning with a tab character then specifies the command needed to make the target.

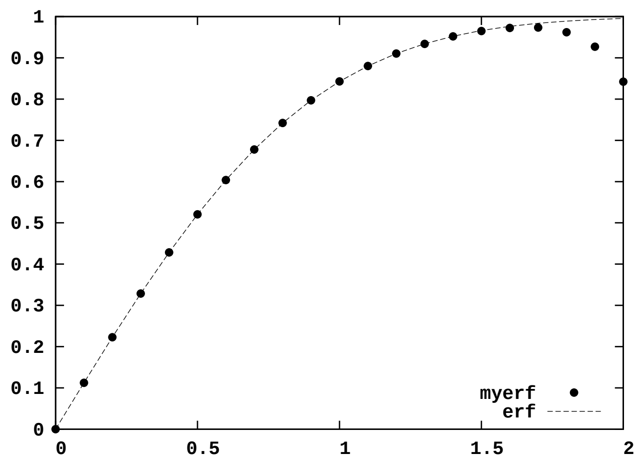
Let's change `main.c` so the loop counts up to 20 and then use `make` to automatically generate a new graph. Edit line 12 so it reads

```
12     for(int i=0;i<=20;i++){
```

Save the file, return to the terminal window and type `make` to apply the rules for updating the graph.

```
netid@hostname:~/lab02$ make
gcc -O2 -o main -std=gnu99 main.c -lm
./main >myerf.dat
gnuplot <myerf.plt
```

Automatically the program was first compiled, then it was run, and finally the plot was generated. If you still have `evince` running, you will notice that the plot now looks like



For larger values of x the filled-in circles representing `myerf` no longer lie on the dashed line representing `erf`. This indicates significant error in our approximation. This can be remedied by taking n larger so that more terms are used in the Taylor expansion; however, if $x = 15$ the function `myerf` gives the wrong answers no matter how large we choose n. This is because the alternating addition and subtraction of large numbers in the Taylor series results in numerical loss of precision when x is large. Since $\operatorname{erf}(x) \approx 1$ when x is large, then it is better to work with $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$ when x is large. We will not pursue this direction of study here.